Natjam: Eviction Policies For Supporting Priorities and Deadlines in Mapreduce Clusters

Brian Cho[1,3], Muntasir Rahman[1], Tej Chajed[1], Indranil Gupta[1], Cristina Abad[1,2], Nathan Roberts[2], and Philbert Lin[1]

[1]University of Illinois at Urbana-Champaign
{bcho2, prlin2, cabad, indy}@illinois.edu
[2]Yahoo! Inc.
nroberts@yahoo-inc.com
[3]Samsung
brian.cho@samsung.com

Abstract

This paper presents Natjam, a system that supports arbitrary job priorities, hard real-time scheduling, and efficient preemption for Mapreduce clusters that are resource-constrained. Our contributions include: i) smart eviction policies for jobs and for tasks, based on resource usage, task runtime, and job deadlines; and ii) a work-conserving task preemption mechanism. We incorporated Natjam into the Hadoop YARN scheduler framework (in Hadoop 0.23). We present experiments from deployments on a test cluster, Emulab and a Yahoo! commercial cluster, using both synthetic traces as well as Hadoop cluster traces we obtained from Yahoo!. Our results reveal that Natjam incurs overheads of under 7%. Under real Hadoop workloads, Natjam performs better than existing techniques.

1 Introduction

Today, computation clusters running engines such as Apache Hadoop [15, 21], Dryad Linq [53], DOT [24], Hive [49], and Pig Latin [37] are used to process a variety of big datasets. The batch Mapreduce jobs in these clusters typically fall into two classes – higher priority jobs and lower priority jobs. Since this *dual priority* setting is common, we call the high priority jobs as *production jobs* and the low priority ones as *research jobs*. For instance, a production job may process click-through logs and decide which ads have reached their advertiser target, and which ads are good alternatives to show. For such jobs, it is critical to produce timely results, since it directly affects revenue. In fact, some production jobs have hard real-time deadlines. On the other hand, a research job may, for instance, identify more lucrative ad placement patterns via a machine learning algorithm on long-term historical click data. Research jobs affect revenue indirectly and therefore they need to complete quickly, but they must be treated at a lower priority than production jobs.

A popular approach today among organizations running their own infrastructure is to provision two physically-separate clusters, one for production jobs and one for research jobs. Administrators tightly restrict the workloads that are allowed on the production cluster, perform admission control manually based on deadlines, keep track of deadline violations via alert systems such as pagers, and subsequently readjust job and cluster parameters manually.

Besides the intensive human involvement, the approach above suffers from: i) long job completion times, and ii) inefficient resource utilization. For instance, jobs in an overloaded production cluster might take longer, even though the research cluster is underutilized (and vice-versa). In fact Mapreduce cluster workloads are time-varying and unpredictable, e.g., in the Yahoo! Hadoop traces we use in this paper, hourly job arrival rates exhibited a max-min ratio as high as 30. Thus, there are times when the cluster is resource-constrained, i.e., it has finite resources compared to incoming demand. Since physically separate clusters cannot reclaim resources from each other, the infrastructure's overall resource utilization stays sub-optimal.

This paper addresses the goal of running a consolidated Mapreduce cluster that supports all jobs, regardless of their priority or deadline. The benefits are high cluster resource utilization, reduced capital costs and, as we show, satisfactory completion times for all jobs.

Focusing on the dual priority setting, our first challenge is that we wish production jobs to finish quickly, but not at the expense of extending many research jobs' completion times. This means that smart strategies are required in selecting which research jobs and constituent tasks are affected by arriving production jobs.

The second challenge is how to dynamically manage the cluster capacity between production and research jobs. A unified scheduler could maintain separate queues (containing jobs) for each priority class and scale up the fraction of the cluster allocated to each queue based on demand [22]. However, scaling down the research queue's capacity requires waiting for some research tasks to finish, which delays production jobs. One could instead kill tasks of research jobs [10, 23], but this entails repeating their work and thus prolongs research jobs. In the deadline-based setting with multiple job priority levels, all jobs belong to one queue, thus waiting for or killing tasks may violate deadlines of their jobs. This means that we need to implicitly and fluidly manage the resources across different job priorities, and in a work-conserving manner.

We present Natjam, a system that provides support for prioritized scheduling of production jobs over research jobs, as well as for production jobs with hard real-time deadlines. The technical contributions of this paper are:

- *Eviction policies for Jobs and Tasks*: When a production job arrives into an occupied cluster, one or more of the research jobs' tasks will need to be preempted to free up resources. This is done by first selecting a victim job (job eviction policy) and then within that job, one or more victim tasks (task eviction policy). We present eviction policies sensitive to: i) resources utilized by a job, and ii) time remaining in a task.
- *Deadline-based scheduling:* For settings with job deadlines, we explore eviction policies that are based on deadlines and on laxity. Laxity accounts for both a job's deadline and its resource usage.
- *Cheap on-demand checkpointing*: Natjam is work-conserving, and it uses a low-overhead suspend and resume mechanism. When a task (of a research or large-deadline job) is suspended, an on-demand checkpoint is created. When the task resumes later it can utilize this saved state to avoid repeating work.
- We incorporate priority and deadline scheduling into the Hadoop YARN scheduler (Hadoop 0.23).

We discuss related work in Section 10, but briefly discuss at a high level how our work is placed. Our focus is on batch jobs rather than streaming or interactive workloads [2, 12, 13, 40, 46, 47]. Some systems have looked at preemption in Mapreduce [6], intelligent killing of tasks [10] (including the Hadoop Fair Scheduler [23]), and SLOs (Service Level Objectives) in generic cluster management [1, 30, 44]. In comparison our work is the first to study the effect of eviction policies and deadline-based scheduling for resource-constrained Mapreduce clusters. Our strategies can be applied orthogonally in systems like Amoeba [6]. We are also the first to incorporate such support directly into Hadoop YARN. Finally, Mapreduce deadline scheduling has been studied in infinite clusters [18, 39, 50, 52] but not in finite clusters.

In this paper, we first discuss eviction policies for the dual priority setting (production and research jobs), and then the design and implementation of the core Natjam system. Next we present an extension, Natjam-R, that builds on Natjam and supports hard time deadlines for production jobs. Our experiments use both synthetic workloads and Hadoop workloads from Yahoo! Inc. We present results from deployments on a test cluster, on Emulab and on a commercial cluster at Yahoo!. Our results reveal that in the dual priority setting, Natjam incurs overheads of under 7% for production and research jobs. Natjam-R meets deadlines with only 20% extra laxity in the deadline compared to the job runtime. We show that under real Hadoop workloads, Natjam is more preferable than existing approaches. We also evaluate Natjam's job and task eviction policies, and draw conclusions about which choices yield the best performance.

2 Eviction Policies for Prioritized Jobs

In the first part of the paper we address the dual priority setting. When a production Mapreduce job arrives at a constrained cluster and there are insufficient resources to schedule it, some tasks of research Mapreduce jobs need to be preempted. Our goals here are to minimize job completion times both for production and for research jobs. This section addresses the twin questions of: 1) How is a victim job chosen so that some of its tasks can be preempted, and 2) Within a given victim job, how are victim task(s) chosen for preemption. We call these as *job eviction* and *task eviction* policies respectively.

The job and task eviction policies are applied in tandem, i.e., for each required task of the arriving production job, a running research task is evicted by applying the job eviction policy first followed by the task eviction policy. This means that a victim job may be evicted partially, i.e., some of its tasks may continue running, e.g., if the arriving job is relatively smaller, or if the eviction policy also picks other victim research jobs.

A quick refresher on Mapreduce is relevant here [15, 21]. A Mapreduce job consists of two phases – map and reduce. Each phase contains multiple parallel tasks. Map tasks are embarrassingly parallel, and each outputs key-value pairs to the reduce phase. A reduce task processes an assigned batch of keys. Map input and reduce output are both stored on the HDFS distributed file system. A job completes only when all its reduce tasks finish.

2.1 Job Eviction Policies

The choice of victim job influences completion time of lower priority research jobs by affecting resources already allocated to them. Thus job eviction policies need to be sensitive to current resource usage. We discuss three resource-aware job eviction policies.

Most Resources (MR): This policy chooses as victim the research job that is currently using the most resources inside the cluster. In Hadoop YARN, this means the number of containers used by the job,[1] while in other versions of Mapreduce this refers to the number of cluster slots. The approach is also extensible to finer-grained notions of resources.

The MR policy, loosely akin to worst-fit policy in OS segmentation, is motivated by the need to evict as few research jobs as possible – a large research job may contain sufficient resources to accommodate one large production job or multiple small production jobs. Thus, fewer research jobs are deferred, more of them complete earlier, and average research job completion time is minimized.

The downside of the MR policy is that when there is one large research job (as might be the case with heavy tailed distributions), it is always victimized whenever a production job arrives. This may lead to starvation and thus a longer completion time for large research jobs.

Least Resources (LR): In order to prevent starving large research jobs, this policy chooses as victim that research job which is currently using the least resources inside the cluster. The reasoning here is that small research jobs which are preempted can always find resources if the cluster frees up even a little in the future. However, the LR policy can cause starvation for small research jobs if the cluster stays overloaded, e.g., if a new production job arrives whenever one completes, LR will pick the same smallest jobs for eviction each time.

Probabilistically-weighted on Resources (PR): In order to address the starvation issues of LR and MR, our third policy called PR selects a victim job using a probabilistic metric based on resource usage. In PR, the probability of choosing a job as a victim is directly proportional to the resources it currently holds. Effectively, PR treats all tasks identically for eviction, i.e., if the task eviction policy were random, the chance of eviction for each task is identical and independent of its job. The downside of PR is that it spreads out evictions across multiple jobs, thus unlike MR, one incoming production job may slow down multiple research jobs.

The latter half of this paper compares these job eviction policies experimentally.

2.2 Task Eviction Policies

Once a victim job has been selected, the task eviction policy is applied within that job to select one task that will be preempted (i.e., suspended). We only consider reduce tasks for eviction – Section 3.1 justifies why. A Mapreduce research job's completion time is determined by its last finishing reduce task. A long tail, or even a single task that finishes late, will extend research job completion time. This concern implies that tasks with shorter remaining time (for execution) must be evicted first. However multiprocessor shortest-task-first scheduling is known to be optimal [48] – in our context this means that the task with the longer remaining time must be evicted first. This motivates two contrasting task eviction policies.

Shortest Remaining Time (SRT): Under this policy, tasks that have the shortest remaining time are selected to be suspended. This policy aims to minimize the impact on the tail of a research job. Further, a task suspended by SRT will finish quickly once it has been resumed. Thus SRT is loosely akin to the longest-task first strategy in multiprocessor scheduling. Rather counter-intuitively, SRT is sometimes provably optimal:

Theorem 2.1: Consider a system where a production job arrival affects exactly one victim job, and evicts several tasks from it. If all these evicted tasks are resumed simultaneously in the future, and we ignore speculative execution, then the SRT eviction policy results in an optimal (lowest) completion time for that research job.

Proof: No alternative policy can do better than SRT in two sub-metrics: i) sum of time remaining for evicted tasks, and ii) tail (i.e., max) of time remaining among the evicted tasks. Thus, when the evicted tasks resume simultaneously, an alternative eviction policy can do only as well as SRT in terms of completion time of the research job. □

We note that the assumption, that tasks of the victim job are resumed simultaneously, is reasonable in those real-life scenarios where production job submission times and sizes cannot be predicted.

Longest Remaining Time (LRT): In this policy, the task with the longest remaining time is chosen to be suspended earlier. This policy is loosely akin to shortest-task first scheduling in multiprocessors. Its main advantage over SRT is that it is less selfish and frees up more resources earlier. LRT might thus be useful in scenarios where production job arrivals are bursty. Consider a victim job containing two tasks – one short and one with

[1] To avoid fragmentation our containers are equi-sized.

a long remaining time. SRT evicts the the shorter task, freeing up resources for one production task. LRT evicts the longer task, but the shorter unevicted task will finish soon anyway, thus releasing resources for *two* production tasks, while incurring only one task suspend overhead. However, LRT can lengthen the tail of the research job, increasing its completion time.

The latter half of this paper compares these task eviction policies experimentally.

3 Natjam Architecture

In order to understand the design decisions required to incorporate eviction policies into a Mapreduce cluster management system, we built Natjam into the popular Hadoop YARN framework in Hadoop 0.23. We now describe Natjam's architecture, focusing on the dual priority setting with production jobs and research jobs.

3.1 Preemption in Hadoop YARN

Background – Hadoop YARN Architecture: In the Hadoop YARN architecture, a single cluster-wide Resource Manager (RM) performs resource management. It is assisted by one Node Manager (NM) per node (server). The RM receives periodic heartbeats from each NM containing status updates about resource usage and availability at that node. The RM runs the Hadoop Capacity Scheduler. The Capacity Scheduler maintains multiple queues which contain jobs. An incoming job is submitted to one of these queues. An administrator can configure two capacities per queue – a fixed capacity and a maximum capacity.

The basic unit of resource allocation for a task is called a *container*. A container is effectively a resource slot that contains sufficient resources (primarily memory) to run one task – either a map, or a reduce, or a master task. An example master task is the Application Master (AM), which is allocated one container. One AM is assigned to each Mapreduce job, and performs job management functions.

An AM requests and receives, from the RM, container allocations needed for its tasks. The AM assigns map and reduce tasks to each container it receives, sends launch requests to the container's NM, and performs speculative execution.

Further an AM sends heartbeats to the RM. The AM also receives periodic heartbeats from its tasks. YARN piggybacks control traffic (e.g., container requests, task assignments) atop heartbeats.

Natjam Components: Natjam entails changes to the Hadoop Capacity Scheduler and the AM, while the NM

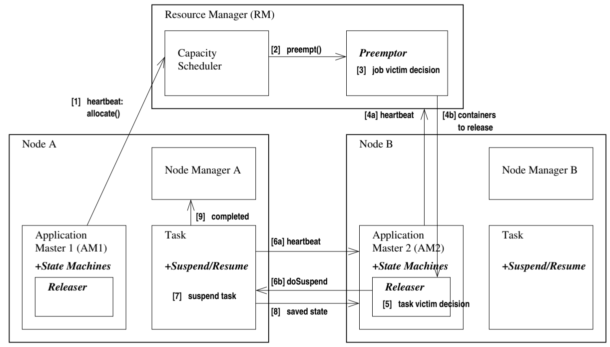


Figure 1: **Example: Container Suspend in Natjam.** *New components are shown in bold font; others are from YARN. AM1 is a production job, AM2 is a research job.*

stays unchanged. Natjam adds the following new components to Hadoop YARN:

1. Preemptor: The preemptor runs as part of the RM. We configure the Capacity Scheduler to contain two queues – one for production jobs and one for research jobs. The preemptor makes preemption decisions by using job eviction policies.

2. Releaser: As part of the AM, this component runs the task eviction policies.

We will detail these in Section 3.2. We now focus on an example of preemption, and then the checkpoint.

Natjam's Preemption Mechanism – Example: Fig. 1 illustrates Natjam's preemption mechanism in YARN. In this example, a research Job 2 is initially executing and fully occupying a cluster, when a production Job 1 requires a single container.[2] In the steps shown, Natjam leverages YARN's heartbeats for efficiency.

Step 1: On AM1's heartbeat, it asks the RM to allocate one container.

Steps 2, 3: The cluster is full, so RM applies the job eviction policies and selects Job 2 as victim.

Step 4: The Preemptor waits for AM2's next heartbeat, and in response sends AM2 the number and type of containers to be released.

Step 5: The Releaser at AM2 uses the task eviction policy to select a victim task.

Step 6: When the victim task (still running) sends its next heartbeat to AM2, it is asked to suspend.

Step 7: The victim task suspends and saves a checkpoint.

Step 8: The victim task sends checkpoint to AM2.

Step 9: The task indicates to NM-A that it has completed and it exits, freeing the container.

This ends the Natjam-specific steps. For completeness, we list below the remaining steps taken by default YARN to give AM1 the new container.

Step 10: NM-A's heartbeat sends container to RM.

Step 11: AM1's next RM heartbeat gets container.

[2]For simplicity we assume AM1 already has a container.

Step 12: AM1 sends NM-A task request.
Step 13: NM-A launches the task on the container.

To Preempt: Maps or Reduces? While preemption can be applied to both map and reduce tasks, Natjam focuses on preemption only for reduce tasks. This is the more challenging case because maps execute each input line independently while reduces execute their inputs in batches (i.e., based on the keys). This focus is also motivated by use case studies which revealed that reduces are substantially longer than maps and thus have a bigger effect on the job tail. For instance, in Facebook workloads the median map task time is 19 s while the median reduce task takes 231 s [55]. While 27.1 map containers are freed per second, only 3 (out of 3100) reduce containers are freed per second. Thus a small production job with 30 reduces would wait on average 10 s, and a large job with 3000 reduces waits 1000 s.

Checkpoint Saved and used by Natjam: When Natjam suspends a research job's reduce task, an on-demand checkpoint is saved containing the following items: i) An ordered *list of past suspended container IDs*, one for each attempt, i.e., each time this task was suspended; ii) *Key counter*, i.e., number of keys that have been processed so far; iii) *Reduce input paths*, i.e., local file path; iv) *Hostname* of last suspended attempt: this is useful for preferrably resuming the research task on the same server.

Additionally, Natjam relies on intermediate task data already available via Hadoop [29]. This includes: v) Reduce inputs, stored at a local host, vi) Reduce outputs, stored on HDFS.

Task Suspend: We modify YARN so that the reduce task keeps track of two pieces of state: paths to files in the local filesystem which hold reduce input, and the key counter, i.e., number of keys that have been processed by the reduce function so far. This (and in general Natjam) does not require any changes to the Hadoop application code, or any end-programmer involvement.

When a suspend request is received from the AM, if the reduce task is in the middle of processing a particular key, it first finishes that key. Second it writes the input file paths to a local log file. Third, Hadoop maintains a *partial output file* per reduce attempt, in the HDFS distributed file system. This holds the output so far from that attempt. We name this partial output file so it includes the container id. When a task suspends this partial output file is closed. Finally the reduce compiles its checkpoint and sends this to its AM. Then the reduce task exits.

Task Resume: The Preemptor is in charge of resuming suspended research tasks. On a resume, the task's AM sends the saved checkpoint state as launch parameters to the NM. When selecting a node to resume on, the RM prefers the old node on which the last attempt ran (available from the hostname field in the checkpoint). If the resumed task is assigned to the same old node, the reduce input can be read without network overhead – reduce input files are just read from local disk. If resumed on a different node, the reduce input is assembled from map task outputs, akin to a new task.

Next the reduce task creates a new partial output file in HDFS. It skips over those input keys that the checkpoint's key counter field indicates have already been processed. It starts execution as a normal reduce task.

Commit after Resume: When a previously suspended reduce task finishes, it needs to assemble its partial output. It does so by first finding, in HDFS, all its past partial output files by using the ordered list of past suspended container ids from its checkpoint. It then accumulates their data into output HDFS files named in that order. This order is critical so that the output is indistinguishable from a reduce task that was never suspended.

3.2 Implementation Issues

This section first describes how we modify the AM state machines. We then detail the Preemptor and Releaser. For efficiency, our implementation leverages existing Hadoop mechanisms such as heartbeats.

Application Master's State Machines: For job and task management, Hadoop YARN's AM maintains separate state machines per job, per task, and per task attempt. Natjam does not change the job state machine – we only enabled this state machine to handle the checkpoint. Thus suspend and resume both occur during the *Running* state in this state machine.

We modify the task state machine minorly. When the AM learns that a task attempt has been suspended (from Step 8 in Fig 1), the task state machine goes ahead and creates a new task attempt to resume the task. The task attempt state machine then takes over.

The task attempt state machine is used by YARN to assign the container, set up execution parameters, monitor progress, and commit output. Natjam adds two states to the task attempt state machine, as shown in Fig. 2: *Suspend-Pending* and *Suspended*. The task attempt has a state of Suspend-Pending when it wishes to suspend a task but has not received suspension confirmation from the local task (Steps 5-7 from Fig. 1). The state becomes Suspended when the saved checkpoint is received (Step 8) – this is a terminal state for that task attempt.

The new transitions for suspension in Fig. 2 are:

- S1: AM asks task to suspend, and requests its checkpoint.
- S2: AM receives task checkpoint; saves in task attempt state machine.

A resuming reduce task starts from the *New* state in the task attempt state machine. However, our modified transitions distinguish it from a new (non-resuming) task

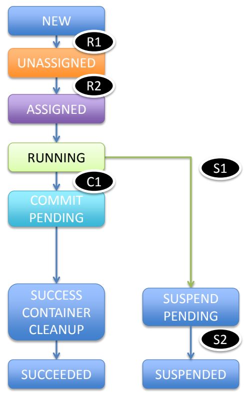


Figure 2: **Modified Task Attempt State Machine:** *At Application Master. Failure states are omitted.*

attempt:

- R1: Like for any reduce task attempt, every heartbeat from the AM to RM requests a container for the resuming reduce. If the RM cannot satisfy the request, it ignores it (the next heartbeat resends it). When resources free up (e.g., when production jobs subside), we make the RM prefer responding to a resuming reduce's request, over one from a non-resuming research reduce. The AM to RM requests also carry the hostname field from the task checkpoint – the RM then prefers container allocation at that hostname.
- R2: Once the AM receives a container from the RM, it launches a task attempt on the allocated container. For resuming reduces, the AM also sends the saved checkpoint to the container.
- C1: On commit, the AM moves output from the partial output files to the final output files in HDFS, as outlined earlier in Section 3.1.

Preemptor: Recall that Natjam sets up the RM's Capacity Scheduler with two queues – one for production jobs and one for research jobs. The Preemptor is implemented as a thread within the Capacity Scheduler. In order to reclaim resources from the research queue for use by the production queue, the Preemptor periodically runs a *reclaim algorithm*, with sleeps of 1 s in between runs. A run generates *reclaim requests*, each of which is sent to some research job's AM to reclaim a container (this is Step 4 in Fig 1). Intuitively, a reclaim request is a production job's intention of acquiring a container.

Initially, we generated a reclaim request whenever: (1) the cluster is full, and (2) the production queue has pending container requests (i.e., requests that have been to the RM, but have not yet been satisfied). However, we discovered that this resulted in a large number of reclaim requests. There is a delay of several seconds between suspension of a container and its subsequent allocation to a job. During this delay the Preemptor ran multiple times and created a duplicate reclaim request each time.

We avoid duplicate reclaim requests by keeping track of a *per-job reclaim list* at the production queue, and deciding when to send reclaim requests based on this list. The reclaim list is maintained as follows: When a reclaim request is sent, it is added to the job's reclaim list. When a container is allocated to that job the oldest reclaim request is removed from the reclaim list. Finally, we changed the second rule for when the Preemptor sends reclaim requests to instead be: ($2'$) the number of pending container requests is greater than the number of requests in the reclaim list.

In extreme cases, the Preemptor may need to kill a container, e.g., if the AM has remained unresponsive for too long. Our threshold to kill a container is when a reclaim request has remained in the reclaim list for longer than a killing timeout (12 s). A kill request is sent directly to the NM to kill the container. This bypasses the AM, ensuring the container will indeed be killed. When a kill request is sent, the reclaim request is now added to an expired list, and remains there for an additional time interval (2 s), when it is assumed the container is dead, and the request is thus removed. With these timeouts, we never observed any tasks killed during any of our runs in any cluster deployment.

Releaser: The Releaser runs at the AM and decides which tasks to suspend. Since the task eviction policies of Section 2.2 (e.g., SRT, LRT) use time remaining at the task, the Releaser needs to estimate this. We use Hadoop's default exponentially smoothed task runtime estimator which relies on the task's observed progress [54]. However, calculating this estimate on-demand can be expensive due to the large numbers of tasks. Thus the AM only periodically estimates the progress of all tasks in the job (once a second), and uses the latest complete set of estimates for task selection. While these might be stale, our experiments show that this approach works well in practice.

Interaction with Speculative Execution: The discussion so far has ignored speculative execution. Hadoop uses speculative execution to replicate straggler task attempts. Natjam does not change speculative execution and works orthogonally, i.e., speculative task attempts are candidates for eviction. When all attempts of a task are evicted, the progress rate calculation of that task is not skewed. This is because speculative execution tracks progress of task attempts rather than tasks themselves. While this interaction could be optimized further, we find that this approach works well under real workloads.

4 Natjam-R: Deadline-based Eviction

We present Natjam-R, an extension of Natjam, targeted at production jobs that have hard and fixed real-time deadlines. While Natjam supported inter-queue preemption (with two queues), Natjam-R adds a layer of *intra-queue* preemption. Thus all jobs can be put into one queue; there is no need for a research queue.

Eviction Policies: Firstly, for job eviction, we explore two deadline-based policies. These are inspired by classical real-time literature [16, 32] and they are *Maximum Deadline First (MDF)* and *Maximum Laxity First (MLF)*. MDF chooses as victim that running job which has the highest deadline. MLF evicts the job with the highest laxity, where *laxity* = *deadline* minus *job's projected completion time*. For MLF we extrapolate Hadoop's reported job progress rate to calculate the job's projected completion time.

While MDF is a static scheduling policy that accounts only for deadlines, MLF is a dynamic policy that also accounts for a job's resource needs. MLF may give a job, which has unsatisfactory progress rate, more resources closer to its deadline. It may do so by evicting small jobs with large deadlines. While MLF may run some large-deadline-high-resource jobs, MDF might starve all large-deadline jobs equally. Further, MLF is fair in that it allows many jobs with similar laxities to make simultaneous progress. However, this fairness can be a shortcoming in scenarios with tight deadlines – MLF results in many deadline misses, while MDF would meet at least some deadlines. Section 7 expands on this issue.

Our task eviction policies remain the same as before (SRT, LRT). This is because the deadline is for the job, not for individual tasks.

In addition to the job and task eviction policies, we need to have a job selection policy. When resources free up, this policy selects a job from among suspended ones and gives it containers. Possible job selection policies are Earliest deadline first (EDF) and Least laxity first (LLF). In fact we implemented these, but discovered thrashing-like scheduling behavior if the job eviction policy was inconsistent with the new job selection policy. For instance, if we used MDF job eviction and LLF new job selection, a job selected for eviction by MDF would soon after be selected for resumption by LLF. We concluded that the new job selection policy needed to be dictated by the job eviction policy, i.e., MDF job eviction implies EDF new job selection, and MLF implies LLF.

Implementation: The main changes in Natjam-R compared to Natjam are in the RM (Resource Manager). The RM now keeps one Capacity Scheduler queue sorted by priority. For MDF and MLF, the priorities are respectively deadline and laxity. The Preemptor periodically (once a second) examines the queue and selects the first job (say J_i) that still has tasks waiting to be scheduled. Then it considers job eviction candidates from the queue, starting with the lowest priority up to J_i's priority. If it encounters a job that still has allocated resources, it is picked as victim, otherwise no further action is taken. Then the Releaser from Natjam uses the task eviction policy to free a container. Thereafter checkpoints, suspend, and resume work the same way as in Natjam (Section 3).

Job	# Reduces	Avg Time (s)
Research-XL	47	192.3
Research-L	35	193.8
Research-M	23	195.6
Research-S	11	202.6
Production-XL	47	67.2
Production-L	35	67.0
Production-M	23	67.6
Production-S	11	70.4

Figure 3: **Microbenchmark Settings**.

5 Microbenchmarks

Experimental Plan: We present four sets of experiments, increasing in complexity and scale. This section presents microbenchmarking results for a small Natjam cluster. Section 6 evaluates a small Natjam deployment driven by real Hadoop traces. Section 7 evaluates Natjam-R. Finally, Section 8 presents a large Natjam deployment under real Hadoop traces.

Microbenchmark Setup: We first evaluate the core Natjam system that supports a dual priority workload, i.e., research and production jobs without deadlines. Our microbenchmarks address the following questions: i) How beneficial is Natjam over existing techniques? ii) What is the overhead of the Natjam suspend mechanism? iii) What are the best job eviction and task eviction policies?

Our test cluster had 7 servers running on a 1 GigE network. Each server had two quad-core processors and 16 GB of RAM, of which 8 GB were configured to run 1 GB-sized Hadoop containers (thus 48 containers were available in the cluster). One server acted as Resource Manager while the other six were workers. Each entity (AM, map task, and reduce task) used one container.

Our experiments inject a mix of research and production jobs, as shown in Fig. 3. To mimic use case studies [55], each job had a small map execution time, and was dominated by reduce execution time. To model variance in task running times, reduce task lengths were selected uniformly from the interval $(0.5, 1.0]$, where 1.0 is the normalized largest reduce task. To emulate com-

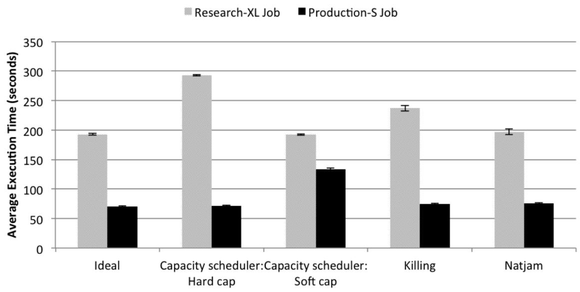


Figure 4: **Natjam vs. Existing Techniques.** *At t=0s Research-XL job submitted; at t=50s Production-S job submitted.*

putations, we used SWIM [9] to create random keys and values, with thread sleeps called in between keys. Shuffle and HDFS traffic were incurred as usual.

The primary metric is job completion time. Each of our datapoints show an average and standard deviation over five runs. Unless otherwise noted, Natjam used MR job eviction and SRT task eviction policies.

Natjam vs. Existing Techniques: Fig. 4 compares Natjam against several alternative settings: i) vs. an ideal setting, ii) vs. two existing mechanisms in the Hadoop Capacity scheduler, and iii) vs. pessimistically killing tasks instead of saving the cheap checkpoint. The ideal setting (i) measures each job's completion time when it is executed on an otherwise empty cluster; thus it ignores resource sharing and context switch overheads. For (ii) we chose the Hadoop Capacity Scheduler because it represents approaches that we might take with two physically-separate clusters sharing the same scheduler. Finally, killing of tasks (iii) is akin to approaches like [10] and the Hadoop Fair Scheduler [23].

A Production-S job was submitted 50s after a Research-XL job. Killing tasks might be attractive because it avoids the overhead of checkpointing. However, the repeated work at each task restart dominates – killing tasks prolongs research jobs by 23% compared to the ideal. Production jobs are unaffected, as expected.

The two Hadoop Capacity Scheduler settings – called *Hard cap* and *Soft cap* – are configured with the research queue set to 75% capacity (36 containers) and production queue to 25% capacity (12 containers) – these capacities gave the best performance. In the Hard cap variant, these capacities are provided as a hard limit for both the research queue and the production queue. In the Soft cap approach, both queues are allowed to expand to the full cluster if there are unused resources, but cannot scale down without waiting for the scheduled tasks to finish. We observe that with Hard cap, the research job takes 52% longer than ideal, while the production job stays unaffected. Under Soft cap, the production job can obtain containers only when the research job frees them – this results in a 85% increase in the production job completion time, while the research job stays unaffected.

The last pair of bars shows that when using Natjam, the production job's completion time is 7% worse (5.4 s) than ideal, and 77% better than Hadoop Capacity Scheduler Soft cap. The research job's completion time is only 2% worse (4.7 s) than ideal, 20% better than that of Killing, and 49% better than Hadoop Hard cap. We study the reasons for this performance benefit in the next paragraphs.

Suspend overhead: We measured Natjam's suspend overhead on a fully loaded cluster. We observed that it took an average of 1.35 s to suspend a task and 3.88 s to resume a task. Standard deviations were low. In comparison, Hadoop took an average 2.63 s to schedule a task on an empty cluster. From this it might appear that Natjam incurs a total overhead of 5.23 s per task attempt. However, in practice the effective overhead is lower – for instance, Fig. 4 showed only a 4.7 s increase in research job completion time. This is because typically task suspends occur in parallel and in some cases task resumes do too. Thus these time overheads are parallelized rather than aggregated.

Task eviction policies: We now compare the two task eviction policies (SRT, LRT) from Section 2.2 against each other, and against a random eviction strategy that we also implemented. We performed two sets of experiments, one with Production-S and another with Production-L. This production job was injected 50 s after the Research-XL job.

Fig. 5 tabulates the results. In all cases the production job incurred similar overhead compared to an empty cluster. Thus we discuss only research job completion time (last column). In the top half of the table, a random task eviction strategy results in a 45 s increase in completion time compared to ideal – we observed that a fourth of the tasks were suspended, leading to a long job tail. Evicting the longest remaining task (LRT) incurs a higher increase of 55 s – once again, this is because LRT prolongs the tail. Evicting the shortest remaining task (SRT) emerges as the best policy and is only 4.7 s worse than ideal. This is because SRT respects the job tail.

In the lower half of the table, a larger production job causes more suspensions. The research job completion times by the random and LRT eviction policies are similar to the top half – this is because its tail was already long with a small production job, and does not grow much for this case. SRT is worse than with a small production job, yet it outperforms the other two eviction strategies.

We conclude that SRT is the best task eviction policy, especially when production jobs are smaller than research jobs. We believe this is a significant use case since research jobs run for longer periods of time and over

Task Eviction Policy	Production Job	Mean (s.d.) run time, in s	Research Job	Mean (s.d.) run time, in s
Random	Production-S	76.6 (3.0)	Research-XL	237.6 (7.8)
LRT	Production-S	78.8 (1.8)	Research-XL	247.2 (6.3)
SRT	Production-S	75.6 (1.5)	Research-XL	197.0 (5.1)
Random	Production-L	75.0 (1.9)	Research-XL	244.2 (5.6)
LRT	Production-L	75.8 (0.4)	Research-XL	246.6 (6.8)
SRT	Production-L	74.2 (1.9)	Research-XL	234.6 (3.4)

Figure 5: **Task Eviction Policies:** *At t=0s, a Research-XL job is submitted; at t=50s the production job is submitted. Job completion times are shown. The ideal job completion times are 192.3 s for research and and 70.4 s for production.*

Job Eviction Policy	Research Job	Mean (s.d.) run time, in s	Research Job	Mean (s.d.) run time, in s
PR	Research-M	195.8 (1.3)	Research-M	201.2 (0.8)
MR	Research-M	196.2 (1.3)	Research-M	200.6 (2.1)
LR	Research-M	200.6 (1.3)	Research-M	228.8 (12.7)
PR	Research-L	201.6 (8.3)	Research-S	213.8 (18.8)
MR	Research-L	195.8 (1.1)	Research-S	204.8 (2.2)
LR	Research-L	195.8 (0.4)	Research-S	252.4 (9.3)

Figure 6: **Job Eviction Policies:** *At t=0s two research jobs are submitted (two Research-M's, or Research-S and Research-L); at t=50s Production-S job submitted. Only research job completion times shown.*

more data, while production jobs are typically small due to the need for faster results.

Job eviction policies: We next compare the three job eviction policies of Section 2.1. Based on the previous results, we always used SRT task eviction. We initially submitted two research jobs, and 50 s later a small production job. We examine two settings – one where the initial research jobs are comparable in size and another where they are different. We observed the production job completion time was close to ideal so we only show the research job completion times in Fig. 6.

The top half of Fig. 6 shows that when research job sizes are comparable, probabilistically weighing job evictions by resources (PR) and evicting the job with the most resources (MR) perform comparably – research job completion times stay within 2 s (0.5%) of each other. This is desirable due to the matching job sizes. However, evicting the job with the least resources (LR) performs worst because it causes starvation in one of the jobs – once tasks start getting evicted from a research job (which may be picked randomly by LR at first if all jobs have the same resource usage), LR will subsequently always pick that job for eviction until it is fully suspended.

This behavior of LR is even more pronounced on small research jobs when research jobs have varying sizes, as in the bottom half of Fig. 6. The Research-S job is picked as victim by PR less often than by LR, and thus PR outperforms LR. PR penalizes the Research-L job slightly more than LR since PR evicts more tasks from a larger job. Even so, PR and MR are within 10 s (5%) of each other – any differences are due to the variable task lengths, and the effectiveness of the SRT task eviction policy. We observed that MR evicted no tasks at all from the Research-S job.

We conclude that when using the best task eviction policy (SRT), the PR and MR job eviction policies are more preferable over LR, with MR especially good under variable research job sizes.

6 Small-scale Deployment

This section presents experiments that answer the following questions under real Hadoop workloads for the dual priority setting: i) What is Natjam's realistic benefit? ii) How does Natjam compare to the Hadoop Capacity scheduler Soft cap, and to Killing of tasks? iii) How does Natjam impact production job completion times?

Setup: We obtained traces from two of Yahoo's Hadoop clusters containing several hundreds of servers. While we cannot reveal details of these traces, we discuss them briefly. The traces cover thousands of job submissions over several hours. They include job submission times, job sizes, number of maps, number of reduces, and other information. The jobs are of different sizes, the arrivals are bursty, and the load varies over time – thus this trace captures a realistic mix of conditions. We injected 1 hour of traces with about 400 jobs into our 7-server test cluster (Section 5) configured with 72 containers (12 for each worker). Natjam used MR job eviction and SRT task eviction.

Since the traces are from a larger cluster, we scaled down the number of tasks in each job to make the workload overload the target cluster. We used different scaling factors for the production and research traces. The production scaling factor was chosen to prevent the production jobs from just overloading the cluster at its peak. The research job scaling factor was chosen so the clus-

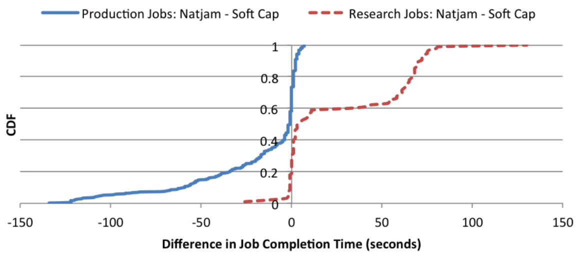


Figure 7: **Small Deployment: Natjam vs. Soft Cap.** *Negative values imply Natjam is better.*

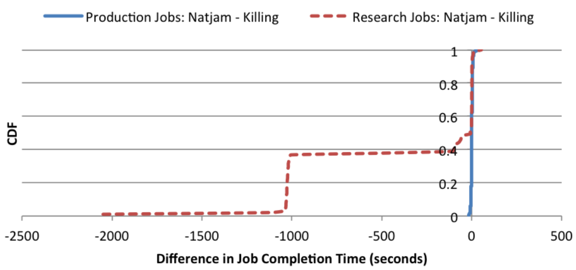


Figure 8: **Small Deployment: Natjam vs. Killing.**

ter was overloaded. Jobs were then submitted at the time indicated by the trace. The tasks were emulated using SWIM [9], incurring the usual shuffle and HDFS traffic.

Natjam's main goals are fast job completion times and high resource utilization. Our approach with scaling factors allows us to hold the cluster utilization high, and thus use job completion time as the main metric.

The results are depicted in Fig. 7 to 9. Each plot compares Natjam vs. an alternative strategy A. We calculated, for each job j_i the quantity = *(Completion time for* j_i *in the Natjam cluster)* minus *(Completion time for* j_i *in the A cluster)*. We then plot the CDF of this quantity. Negative values on the CDF imply Natjam completes the job earlier than the alternative strategy.

Natjam vs. Soft Cap: We configured the Hadoop Capacity Scheduler Soft cap with 80% capacity for production and 20% for research jobs – recall that this mechanism allows the allocated capacities for each class to scale up. Fig. 7 shows that compared to the Soft cap approach, with Natjam 40% of production jobs finish at least 5 s earlier, 20% finish 35 s earlier, and 10% finish 60 s earlier. Only 3% of jobs perform 5 s or worse with Natjam. In fact, 60% of research jobs are delayed 30 s or less. We believe this is a reasonable tradeoff to accommodate production jobs.

Natjam vs. Killing: Fig. 8 shows that compared to the cheaper approach of killing research jobs, saving their checkpoint in Natjam tremendously improves research job completion times. 36% of research jobs finish at least 1000 s earlier. In fact we observed that in the Killing setting, many research jobs finished well after the last job

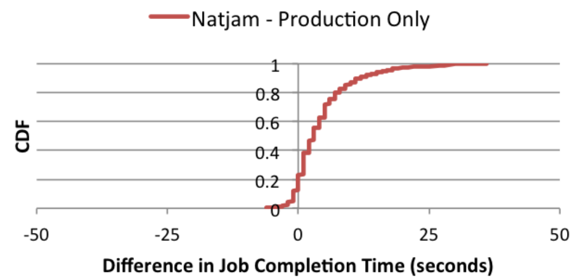


Figure 9: **Small Deployment: Natjam vs. Production Only.**

was submitted.

Natjam does not affect production jobs much, compared to Killing. The mean and median are within a second of each other, and the absolute decrease and increase in performance at the 1st percentile and 99th percentile are within 2 s. We conclude that under realistic workloads, checkpointing is still preferable to killing tasks.

Natjam vs. Production Only: To evaluate Natjam's effect on production jobs, we compared two clusters – a Natjam cluster receiving the production + research trace as above, and a Hadoop cluster (Soft Cap) receiving only the production job part of the trace (labeled Production Only). Fig. 9 shows that for production jobs, the median Natjam job is within 3 s of Production Only's median, while the mean is within 4.5 s of Production Only.

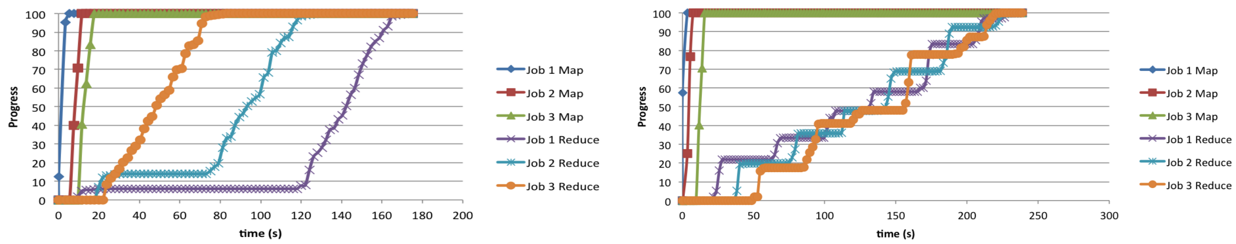
The maximum difference in completion time is 36 s. This value is high due to two factors. First is Natjam's overhead for suspending tasks (Section 5). Second is an implementation bottleneck arising out of Natjam's Hadoop integration where concurrent requests for starting AMs are serialized. These factors were amplified because of the small cluster size – they disappear in Section 8 where a large cluster leads to a higher rate of containers becoming free.

7 Natjam-R Evaluation

We now evaluate the real-time support of our Natjam-R extension (Section 4). Our experiments address the following questions: i) How do MDF and MLF job eviction strategies compare? ii) How good is Natjam-R at meeting deadlines? iii) Do Natjam-R's benefits hold under realistic Hadoop workloads?

For diversity, we use a different target cluster in this section. We use 8 Emulab servers [17], each with 8 core Xeon processors and 250 GB disk space. One of the servers is the Resource Manager, and each of the other seven servers runs 3 containers of 1 GB each (thus 21 containers total).

MDF vs MLF: We injected three identical jobs, Job 1 to Job 3, each with 8 maps and 50 reduces (each job took 87



(a) (b)

Figure 10: **Natjam-R: (a) MDF vs. (b) MLF.** *Lower index jobs have lower deadline but arrive later.*

s on an empty cluster). They were submitted in that order starting at t=0 s and 5 s apart, thus overloading the cluster. Since MDF and MLF will both meet lax deadlines, we chose tight deadlines. The deadlines of Job 1 to Job 3 were set as 200 s, 190 s and 180 s respectively – this order forced preemption. Our conclusions would apply to different deadlines as long as they were 10 s apart.

Fig. 10 depicts the progress rate for the MDF cluster and the MLF cluster. Our first observation is that while MDF allows the small deadline jobs to run earlier and thus satisfy all deadlines, MLF misses all deadlines in Fig. 10b. In MLF, jobs proceed in lockstep after a while in the reduce phase – this occurs because when a lower laxity job (e.g., Job3) has run for a while in lieu of a higher laxity job (e.g., Job1), their laxities become comparable. Thereafter, each job alternately preempts the other. Breaking ties, e.g., by using deadline, does not eliminate this behavior. In a sense, MLF tries to be fair to all jobs allowing them all to make progress simultaneously, but this fairness is in fact a drawback.

Secondly, MLF takes longer to finish all three jobs, i.e, 239 s compared to MDF's 175 s. This indicates that MLF's lockstep behavior incurs a high context switch overhead. We conclude that MDF is preferable to MLF, especially under tight deadlines.

Varying the Deadline: We submitted a job (Job 1), and 5 s later an identical job (Job 2) with a lower deadline. For each job we measured its *clean compute time* as the time to run the job in an empty cluster. Then, we set its *deadline = submission time + (clean compute time* $\times$ *(1+ε))*. Fig. 11 shows the effect of ε on a metric called *margin*. We define a job's *margin = (deadline)* minus *(job completion time)*. A negative margin implies a deadline miss. We observe that an ε as low as 0.8 still meets both deadlines, while an ε as low as 0.2 misses one deadline (the higher deadline). This means that given one critical job with a very low deadline, Natjam-R can satisfy it if it has at least 20% more time than the job's clean compute time – this percentage estimates the overhead of Natjam-R. We also performed experiments which varied the second job's size as a fraction of the first job from 0.4 to 2.0, but we saw no effect on margin.

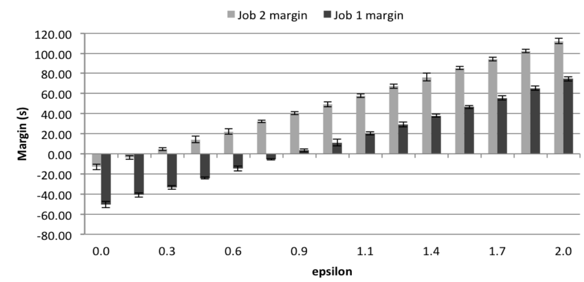


Figure 11: **Natjam-R: Effect of Deadlines***: Margin = Deadline - Job completion time, thus a negative margin implies a deadline miss. Job 2 has a lower deadline and is submitted 5 s after Job 1.*

Trace-Driven Experiments: We used the Yahoo! Hadoop traces earlier from Section 6 to evaluate Natjam-R's deadline satisfaction. We used only the production cluster trace, with a scaling factor selected so as to overload the target cluster. Since the original system did not support deadline scheduling, no deadlines were available from the traces. Thus we chose ε randomly for each job from the interval $[0, 2.0]$, and used this to set its deadline forward from its submission time. A given job's deadline was selected to be the same in all runs.

We compare Natjam-R against Hadoop Soft cap. Fig 12 shows the CDF of the difference in the margins of these two approaches – a negative difference implies Natjam-R is better. Natjam-R's margin is better than Soft cap for 69% of the jobs. The largest improvement in margin was 366 s. The plot is biased by one outlier job that took 1000 s longer in Natjam-R; the second-highest negative difference is only -287 s. This outlier job suffered in Natjam-R because the four jobs submitted just before it and one job right after had much tighter deadlines. In comparison, Soft cap scheduled this outlier job in order. Yet the conclusion is positive – among the 400 jobs with variable deadlines, there was only one such outlier. We conclude that overall, Natjam-R satisfies deadlines well.

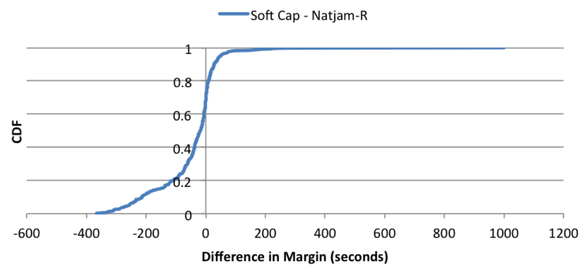


Figure 12: **Natjam-R: Effect of real Yahoo! Hadoop Trace**: *Margin = Deadline - Job completion time. Negative values imply Natjam-R is better.*

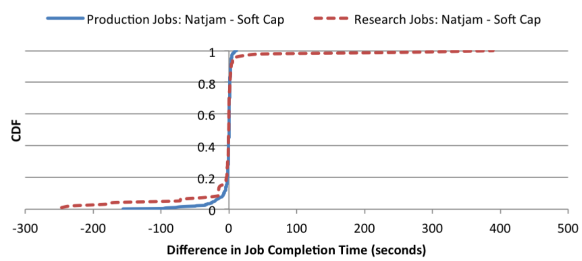


Figure 13: **Large-scale Deployment: Natjam vs. Hadoop Soft Cap.** *Negative values imply Natjam is better.*

8 Large-scale Deployment

Finally, we return to the dual priority Natjam, and evaluate it on a real Yahoo datacenter. We address the same questions as Section 6 but for a larger deployment setting. The target Yahoo datacenter consisted of 250 servers which we configured to run a total of 2000 containers. We used the Yahoo! Hadoop traces of Section 6 with scaling factors for production jobs so as to just fit within the target cluster, and for research jobs so as to overload the target cluster. The total shuffle traffic was measured at about 60 GB, and HDFS incurred 100 GB read and 35 GB write.

The results are shown in Fig. 13 to 15. The trends are similar to those we saw in Section 6 but the distributions are different. This is due to the larger cluster and the resultant higher rate of containers becoming free.

Natjam vs. Soft Cap: Fig. 13 shows that for production jobs, Natjam completes 53% of jobs earlier than Hadoop Soft cap. Further, 12% of these jobs finish at least 5 s earlier than in Soft cap, and fewer than 3% jobs finish 5 s or later. In fact we observe that at the 5th percentile jobs finish 20 s or earlier, at the 2nd percentile 60 s or earlier, and at the 1st percentile 80 s or earlier. The largest improvement over Soft cap is more than 150 s.

For research jobs, Natjam completes 63% of jobs earlier than Soft cap. The top right part of the curve is due to only two outlier jobs that were 260 s and 390 s slower under Natjam. We conclude that compared to Hadoop Soft cap, Natjam improves completion times for a significant fraction of production and research jobs.

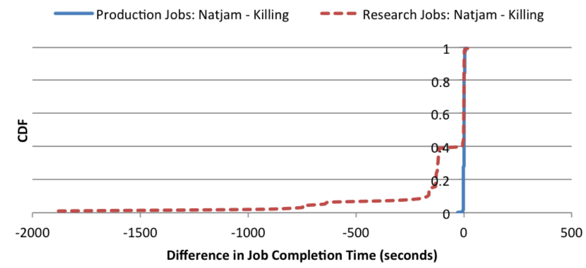


Figure 14: **Large-scale Deployment: Natjam vs. Killing.**

Natjam vs. Killing: Fig. 14 shows that compared to killing research jobs, Natjam's checkpointing improves research job completion times by over 100 s for 38% of jobs, and at the 5th percentile Natjam is almost 750 s faster. The largest improvement observed was 1880 s. Natjam does not affect production jobs much – completion times for the two approaches are within 1 s of each other at the 99th and 1st percentiles.

Natjam vs. Production Only: Fig. 15 shows the median Natjam job completion time is within 40 ms of Production Only, while the mean is within 200 ms. Thus Natjam's checkpointing has minimal impact on production jobs.

We conclude that even in a large datacenter with 250 servers and under a realistic workload, Natjam provides benefits over existing approaches.

9 Discussion

We now discuss possible extensions to our system.

Rack-level Locality: Currently in Natjam, a resumed task can reuse reduce input files only when it is resumed on the same server as its last task attempt. Otherwise, network transfers are required from all map tasks. This can be ameliorated by using HDFS to save the reduce checkpoint. Globally accessible reduce input will allow a reduce to resume efficiently on any server. To decide where a task attempt resumes, a multi-level preference order can be used: first prefer the server of the last attempt, then a server in the same rack, and then any server. To lower overhead, HDFS can be modified to store only one replica of the checkpoint. In case of failure, reduce input can be obtained again from map tasks. HDFS can store this at the writing node itself, thus invoking only a local disk write.

Suspending Stateful Reduces: As described so far, Natjam does not require any changes to the user's Hadoop code. However, some Mapreduce tasks have stateful reduce tasks, i.e., the task saves state across

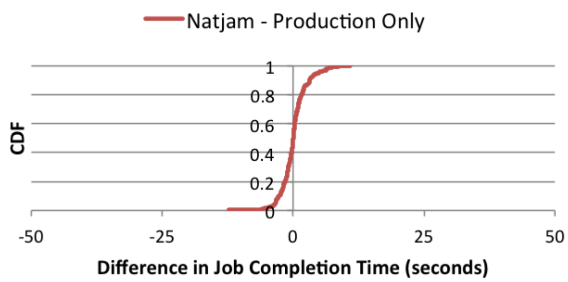


Figure 15: **Large-scale Deployment: Natjam vs. Production Only.**

keys. Natjam can support this via serialize and deserialze methods. When a task is suspended, inter-key state datastructures are serialized and copied to HDFS. When the task resumes, Natjam deserializes the checkpoint and skips to the current key counter. These two methods are application-dependent, hence the Hadoop programmer needs to write them. We believe that in most cases these can be written in such a way as to maintain small checkpoints. For instance, consider [31] which computes relative frequency across word co-occurrences. Reduce input keys are word pairs (w, Σ) so that all $(w, *)$ pairs occur together. Thus for each given w, the reduce maintains a running sum for the $(w, *)$ pairs seen so far. The serialize and deserialize methods would merely maintain an integer field to store this sum.

Chained Jobs: Data analysis frameworks [37, 49] and workflow schedulers [38] create chains or DAGs of Hadoop-like jobs, which might be associated with one deadline. For constrained clusters, Natjam can be used to solve this problem by leveraging critical path-based algorithms [18, 43] to calculate deadlines of constituent Hadoop jobs, and ParaTimer [34] to estimate progress rate.

10 Related Work

OS mechanisms: Sharing finite resources among applications is a fundamental issue in Operating Systems [48]. Not surprisingly, Natjam's eviction policies are analogous to multiprocessor scheduling techniques (e.g., shortest task first), and to eviction policies for caches and paging systems. However, our results are different because Mapreduce jobs need to have all tasks finish. We clarify that Natjam does not rely on priorities at the OS level. This would require tight and prohibitive integration between the OS and Hadoop scheduling. PACman [4] looks at eviction policies for caches in Mapreduce clusters, and it can be used orthogonally with Natjam.

Preemption: Amoeba, a system built in parallel with ours, provides instantaneous fairness with elastic queues, and uses a checkpointing mechanism [6]. The main differences in Natjam compared to Amoeba are: i) we focus on job and task eviction policies, ii) we focus on jobs with hard deadlines, and iii) our implementation works directly with Hadoop 0.23, while Amoeba requires the prototype Sailfish system [41]. Further, Sailfish was built on Hadoop 0.20 – since then, Hadoop 0.23 has addressed many relevant bottlenecks, e.g., using read-ahead seeks, Netty [36] to speed up shuffle, etc. Finally, we wish to note that our eviction and scheduling policies can be implemented orthogonally in Amoeba.

Delay scheduling [55] avoids killing map tasks while achieving data locality. In comparison, Natjam focuses on reduce tasks as they are longer than maps, and they release resources slower – this makes our problem more challenging. Global preemption [10] selects tasks to kill across all jobs. However killing tasks is suboptimal as we showed experimentally.

Compared to our previous short work [35], this paper explores eviction policies and deadlines.

Real-time Scheduling: ARIA [50] and Conductor [52] estimate how a Hadoop job needs to be scaled up to meet to its deadline, e.g., based on profiles of past executions or a constraint satisfaction problem. They do not target clusters with finite resources. Real-time constraint satisfaction problems were solved analytically [39], and Jockey [18] addressed DAGs of data-parallel jobs – however eviction policies or Hadoop integration were not fleshed out. Statistics-driven approaches have been used for cluster management [19] and for Hadoop [28]. Much work has also been done in speeding up Mapreduce environments by tackling stragglers, e.g., [5, 54]. However, these do no not support job priorities and deadlines.

Dynamic proportional share scheduling [42] allows applications to bid for resources, but is driven by economic metrics rather than priorities or deadlines. Data transfer within a data center can be prioritized to speed up time-sensitive jobs [11], and Natjam can be used orthogonally.

Natjam focuses on batch jobs rather than stream processing or interactive queries. Stream processing in the cloud has been looked at intensively, e.g., Hadoop Online [12], Spark [46], Storm [47], Timestream [40] and Infosphere [25]. BlinkDB [2] and MeT [13] optimize interactive queries for SQL and NoSQL systems.

Finally, classical work on real-time system has proposed a variety of scheduling approaches including classical EDF and rate monotonic scheduling [32, 33], priority-based scheduling of periodic tasks [20], laxity-based approaches [16], and handling task DAGs [43] – Natjam is different in that we focus on Mapreduce workloads.

Fairness: Providing fairness across jobs has been a recent focus in cloud computing engines. This includes Hadoop's Capacity Scheduler [22] and Fair Scheduler [23], which provide fairness by allowing an administrator to configure queue capacities and job priorities. Quincy [26] solves an optimization problem to provide fairness in DryadLinq [53]. None of these works allow resource preemption [27]. Finally, there has been recent focus on satisfying SLAs [7] and satisfying real-time QoS [3] but they do not target Mapreduce clusters.

Cluster Management with SLOs: Recently several cluster management systems have targeted SLOs, e.g., Omega [44], Cake [51], Azure [8], Centrifuge [1] and Albatross [14]. Mesos [30] uses dominant resource fairness across applications sharing a cluster, and Pisces [45] looks at multi-tenant fairness in key-value stores.

11 Summary

This paper presented Natjam and Natjam-R, which provide support for dual priorities and hard-real time scheduling, for jobs in constrained Mapreduce clusters. Among several eviction policies we found that the MR (Most Resources) job eviction and SRT (shortest remaining time) task eviction policies are the best for the dual priority setting. For the hard-real time setting, the MDF (maximum deadline first) job eviction policy performed the best. Natjam incurs only a 2-7% context switch overhead. Natjam-R meets deadlines with only 20% extra laxity in deadline. Under real Hadoop workloads and for a variety of cluster sizes, compared to existing techniques, Natjam improves completion times for both production and research jobs, and Natjam-R satisfies more job deadlines.

References

[1] A. Adya, J. Dunagan, and A. Wolman. Centrifuge: Integrated lease management and partitioning for cloud services. In *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[2] S. Agarwal, A. Panda, B. Mozafari, H. Milner, S. Madden, and I. Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proc. ACM European Conference on Computer Systems (Eurosys)*, 2013.

[3] M. Amirijoo, J. Hansson, and S.H. Son. Algorithms for managing QoS for real-time data services using imprecise computation. *Real-Time and Embedded Computing*, pages 136–157, 2004.

[4] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica. PACMan: coordinated memory caching for parallel jobs. In *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[5] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using Mantri. In *Proc. Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[6] G. Anantharanayanan, C. Douglas, R. Ramakrishnan, S. Rao, and I. Stoica. True elasticity in multi-tenant clusters through Amoeba. In *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2012.

[7] A. Andrzejak, D. Kondo, and S. Yi. Decision model for cloud computing under SLA constraints. In *Proc. IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 257–266, 2010.

[8] B. Calder, A. Ogus J. Wang, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. Fahim ul Haq, M. Ikram ul Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, pages 143–157, 2011.

[9] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating Mapreduce performance using workload suites. In *Proc. IEEE Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 390–399, july 2011.

[10] L. Cheng, Qi Q. Zhang, and R. Boutaba. Mitigating the negative impact of preemption on heterogeneous Mapreduce workloads. In *Proc. Conference on Network and Service Management (CNSM)*, pages 189–197, 2011.

[11] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with Orchestra. In *Proc. ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, pages 98–109, 2011.

[12] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sear. Mapreduce online.

In *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[13] F. Cruz, F. Maia, M. Matos, R. Oliveira, J. Paulo, J. Pereira, and R. Vilaca. MeT: workload aware elasticity for NoSQL. In *Proc. ACM European Conference on Computer Systems (Eurosys)*, 2013.

[14] S. Das, S. Nishimura, and D. Agrawal. A. El Abbadi. Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration. In *Proc. Conference on Very Large Data Bases (VLDB)*, pages 494–505, 2010.

[15] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM (CACM)*, 51:107–113, January 2008.

[16] M. L. Dertouzos and A. K. Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on Software Engineering*, 15(12), 1989.

[17] Emulab. `http://emulab.net`.

[18] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: guaranteed job latency in data parallel clusters. In *Proc. ACM European Conference on Computer Systems (EuroSys)*, pages 99–112, 2013.

[19] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the cloud. In *Proc. International Workshop on Self Managing Database Systems (SMDB)*, pages 87–92, 2010.

[20] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-time systems*, 2(3), 2003.

[21] Hadoop. `http://hadoop.apache.org/`.

[22] Hadoop Capacity Scheduler. `http://hadoop.apache.org/docs/stable/capacity_scheduler.html`, 2013.

[23] Fair Scheduler. `http://hadoop.apache.org/docs/stable/fair_scheduler.html`, 2013.

[24] Y. Huai, R. Lee, S. Zhang, C. H. Xia, and X. Zhang. Dot: a matrix model for analyzing, optimizing and deploying software for big data analytics in distributed systems. In *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2011.

[25] IBM Infosphere Platform. `http://www-01.ibm.com/software/data/infosphere/`, 2013.

[26] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proc. of ACM Symposium on Operating Systems Principles (SOSP)*, pages 261–276, 2009.

[27] HADOOP-5726 JIRA: Remove pre-emption from the capacity scheduler code base. `https://issues.apache.org/jira/browse/HADOOP-5726`, 2009.

[28] K. Kambatla, A. Pathak, and H. Pucha. Towards optimizing Hadoop provisioning in the cloud. In *Proc. Usenix Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[29] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making cloud intermediate data fault-tolerant. In *Proc. Symposium on Cloud Computing (SoCC)*, pages 181–192, 2010.

[30] A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[31] J. Lin and C. Dyer. *Data-intensive text processing with Mapreduce*. Morgan & Claypool Publishers, 2010.

[32] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20:46–61, Jan 1993.

[33] J. S. Liu. *Real-time systems*. Prentice Hall, 2000.

[34] K. Morton, M. Balazinska, and D. Grossman. Paratimer: a progress indicator for Mapreduce DAGs. In *Proc. ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 507–518, 2010.

[35] Anonymous, details omitted for double-blind reviewing. 2013.

[36] Netty. `http://netty.io/`, 2013.

[37] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *Proc. of ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1099–1110, 2008.

[38] Oozie. `http://oozie.apache.org/`, 2013.

[39] L. Phan, Z. Zhang, B. Loo, and I. Lee. Real-time MapReduce scheduling. Technical Report MS-CIS-10-32, University of Pennsylvania, Feb 2010.

[40] Z. Qian, Y. He, C. Su Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. Reliable stream computation in the cloud. In *Proc. ACM European Conference on Computer Systems (Eurosys)*, 2013.

[41] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and Damian Reeves. Sailfish: a framework for large scale data processing. In *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2012.

[42] T. Sandholm and K. Lai. Dynamic proportional share scheduling in Hadoop. In *Proc. Conference on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pages 110–131, 2010.

[43] I. Santhoshkumar, G. Manimaran, and C. Siva Ram Murthy. A pre-run-time scheduling algorithm for object-based distributed real-time systems. *Journal of Systems Architecture*, 45(14), 1999.

[44] M. Schwarzkopf, A. Konwinski, M. Abd el Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In *Proc. ACM European Conference on Computer Systems (Eurosys)*, 2013.

[45] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proc. Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, pages 349–362, 2012.

[46] Spark: Lightning-fast cluster computing. `http://spark-project.org/,2013`.

[47] Storm. `http://storm-project.net/`, 2013.

[48] A. S. Tanenbaum. The Operating System as a Resource Manager. In *Modern Operating Systems*, chapter 1.1.2. Pearson Prentice Hall, 3rd edition, 2008.

[49] A. Thusoo, J. Sen Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a Mapreduce framework. *Proc. Very Large Data Base Endowment (PVLDB)*, 2:1626–1629, August 2009.

[50] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: automatic resource inference and allocation for mapreduce environments. In *Proc. International Conference on Autonomic Computing (ICAC)*, 2011.

[51] A. Wang, S. Venkataraman, S. Alspaugh, R. Katz, and I. Stoica. Cake: enabling high-level SLOs on shared storage systems. In *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2012.

[52] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the deployment of computations in the cloud with Conductor. In *Proc. Usenix Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[53] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–14, 2008.

[54] M. Zaharia, A. Konwinski, A.D. Joseph, R. Katz, and I. Stoica. Improving Mapreduce performance in heterogeneous environments. In *Proc. Usenix Symposium on Operating Systems Design and Implementation (OSDI)*, pages 29–42, 2008.

[55] M. Zaharia, D. Borthakur J. Sen Sarma, Khaled K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proc. ACM European Conference on Computer Systems (Eurosys)*, pages 265–278. ACM, 2010.